TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MIMMI MATALAMÄKI

QUICKSORTIN, INSERTION SORTIN JA HEAP SORTIN TOTEUTUSTAPOJEN VERTAILU

Kandidaatintyö

Tarkastaja: Tiina Schafeitel-Tähtinen
Jätetty tarkastettavaksi 02. joulukuuta 2018

TIIVISTELMÄ

MIMMI MATALAMÄKI: Quicksortin, insertion sortin ja heap sortin toteutustapojen vertailu
Tampereen teknillinen yliopisto
Kandidaatintyö, 21 sivua
Joulukuu 2018
Tietotekniikan kandidaatin tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: Tiina Schafeitel-Tähtinen

Avainsanat: algoritmi, järjestysalgoritmi, quicksort, insertion sort, heap sort

Järjestysalgoritmeilla on merkittävä rooli tietotekniikassa, koska monet algoritmit tarvitsevat syötteitään järjestyksessä tai algoritmien tarvitsee järjestää tietoja suorituksensa aikana. Tässä työssä tutkitaan kolmea eri järjestysalgoritmia. Tutkittavat järjestysalgoritmit ovat quicksort, insertion sort ja heap sort. Työn tavoitteena on selvittää mitkä ovat näiden kolmen järjestysalgoritmin suurimpia eroja algoritmien toteutustasolla.

Työ on toteutettu suurimmaksi osaksi kirjallisuustutkimuksena. Kirjallisuustutkimus esittelee jokaisen algoritmin toiminnan ja toteutuksen. Algoritmien asymptoottisen suoritusajan vertailussa on käytetty laadullista tutkimusta, kun on laskettu algoritmien suorittamien operaatioiden määriä parhaimmassa, keskimääräisessä ja pahimmassa tapauksessa.

Tutkimuksessa selvisi, että algoritmien toimintalogiikat eroavat toisistaan huomattavasti. Siinä missä quicksort ja insertion sort lajittelevat saamansa syötteen, heap sort luo syötteestä uuden tietorakenteen (maksimi- tai minimikeon) ja järjestää sen. Vaikka quicksortin ja heap sortin toimintalogiikat eroavat suuresti toisistaan, ovat ne molemmat epävakaita algoritmeja ja insertion sort on vakaa algoritmi. Heap sortin asymptoottinen suoritusaika on aina nlogn ja quicksortin ja insertion sortin asymptoottinen suoritusaika riippuu siitä, onko kyseessä paras, keskimääräinen vai pahin tapaus.

SISÄLLYSLUETTELO

KUVALUETTELO

.

1. JOHDANTO

Algoritmeja on ollut ihmisten käytössä jo tuhansia vuosia. Perusidea algoritmeissa on löytää mahdollisimman hyvä ratkaisu ongelmaan suorittamalla tietty määrä operaatioita. Algoritmeja käytetään nykyään monella eri alalla muun muassa taloudessa, teollisuudessa ja ihmisten välisessä kommunikoinnissa. Monet algoritmit vaativat syötteenä saamansa datan olevan tietyssä järjestyksessä, jotta voivat suorittaa omat operaationsa sille. Data järjestetään käyttäen erilaisia järjestysalgoritmeja. Näin ollen järjestysalgoritmit ovat tärkeässä osassa monien ongelmien ratkaisuissa. [1, luku 1] Tässä työssä keskitytään tutkimaan tarkemmin kolmea eri järjestysalgoritmia: quicksortia, insertion sortia ja heap sortia. Kyseiset algoritmit on valittu, koska ne soveltuvat erilaisiin käyttötarkoituksiin ja ovat kukin toteutettu erilailla. Algoritmien valintakriteerinä oli myös valita algoritmit, jotka ovat laajalti käytössä.

Tässä työssä on tarkoituksena selvittää mitkä ovat quicksortin, insertion sortin ja heap sortin suurimmat erot algoritmien toteutustasolla. Tarkoituksena on ottaa selvää miten kyseisten algoritmien toimintalogiikat eroavat toisistaan. Lisäksi selvitetään mitä eroja algoritmeilla on kertaluokkanotaatioissa, siinä miten suuren tietomäärän käsittelyyn ne soveltuvat, mitä suunnitteluperiaatetta minkäkin algoritmin toteutukseen on käytetty ja onko algoritmit vakaita vai epävakaita. Työ vastaa kysymykseen, mitkä ovat quicksortin, insertion sortin ja heap sortin suurimmat erot algoritmien toteutustasolla. Tässä työssä tutkimustapana käytetään kirjallista selvitystä ja laadullista tutkimusta.

Luvussa 2 esitellään algoritmi yleisesti, käydään läpi algoritmien asymptoottinen suoritusaika sekä miten algoritmeja suunnitellaan. Lisäksi käydään läpi binary-heap -tietorakenne ja sen käyttö lajittelualgoritmissa. Luvussa 3 esitellään vertailtavat järjestysalgoritmit yksitellen. Luku 4 keskittyy luvussa 3 esiteltyjen järjestysalgoritmien vertailuun. Järjestysalgoritmeista vertaillaan ainakin niiden toimintalogiikkaa, asymptoottista suoritusaikaa, minkä suuruiselle tietomäärälle algoritmit sopivat, kuinka yleisesti algoritmit ovat käytössä, mitä suunnitteluperiaatetta noudattaen algoritmit on toteutettu sekä ovatko algoritmit vakaita vai eivät. Luku 5 on yhteenvetoluku.

2. ALGORITMI YLEISESTI

Algoritmi on joukko sääntöjä, joita suoritetaan tietyssä järjestyksessä, jotta saadaan haluttu lopputulos. Algoritmilla tulee olla seuraavat ominaisuudet, jotta se voidaan luokitella algoritmiksi: määrällinen syöte, ainakin yksi tulos ja säännöt, jotka ovat selkeitä ja tarkoituksenmukaisia. Lisäksi algoritmin tulee päättyä joutumatta loputtomaan silmukkaan, jos säännöt on suoritettu oikeassa järjestyksessä. Algoritmeilla on tärkeä rooli ihmisten jokapäiväisessä elämässä sekä monilla tieteenaloilla. Esimerkiksi käytännössä kaikki mitä tietokoneilla tehdään, perustuu joko suorasti tai epäsuorasti algoritmeihin. Algoritmit ovat tärkeässä roolissa ainakin seuraavilla aloilla: tietorakenteet, salaustekniikoiden käyttö, automaattinen tuotanto, biologia sekä elektroniikka. Uusia algoritmeja kehitettään tietorakenteiden alalla. [2, luku 1]

Tässä luvussa käsitellään ensin algoritmien asymptoottinen suoritusaika luvussa 2.1. Luku 2.2 keskittyy algoritmien suunnitteluun ja siinä käydään läpi kaksi algoritmien suunnitteluperiaatetta: hajota ja hallitse –suunnitteluperiaate sekä suunnitteluperiaate, jossa algoritmi hyödyntää saamaansa syötettä. Luvussa 2.3. käsitellään kolmas suunnitteluperiaate, joka on binary-heap tietorakenteen käyttö järjestysalgoritmissa.

2.1 Algoritmien asymptoottinen suoritusaika

Algoritmin asymptoottinen suoritusaika saadaan jättämällä algoritmista sen kasvun kannalta kaikki oleellisin osa. Esimerkiksi algoritmin n^2+50*n+200, missä n on mikä tahansa ei-negatiivinen kokonaisluku, joka kuvaa alkioiden määrää, asymptoottinen suoritusaika saadaan poistamalla termit 50*n ja 200. Algoritmin asymptoottiseksi suoritusajaksi jää n^2, joka on algoritmin kasvun kannalta kaikki oleellisin osa. Asymptoottinen suoritusaika kuvaa siis algoritmin kasvunopeutta. [5]

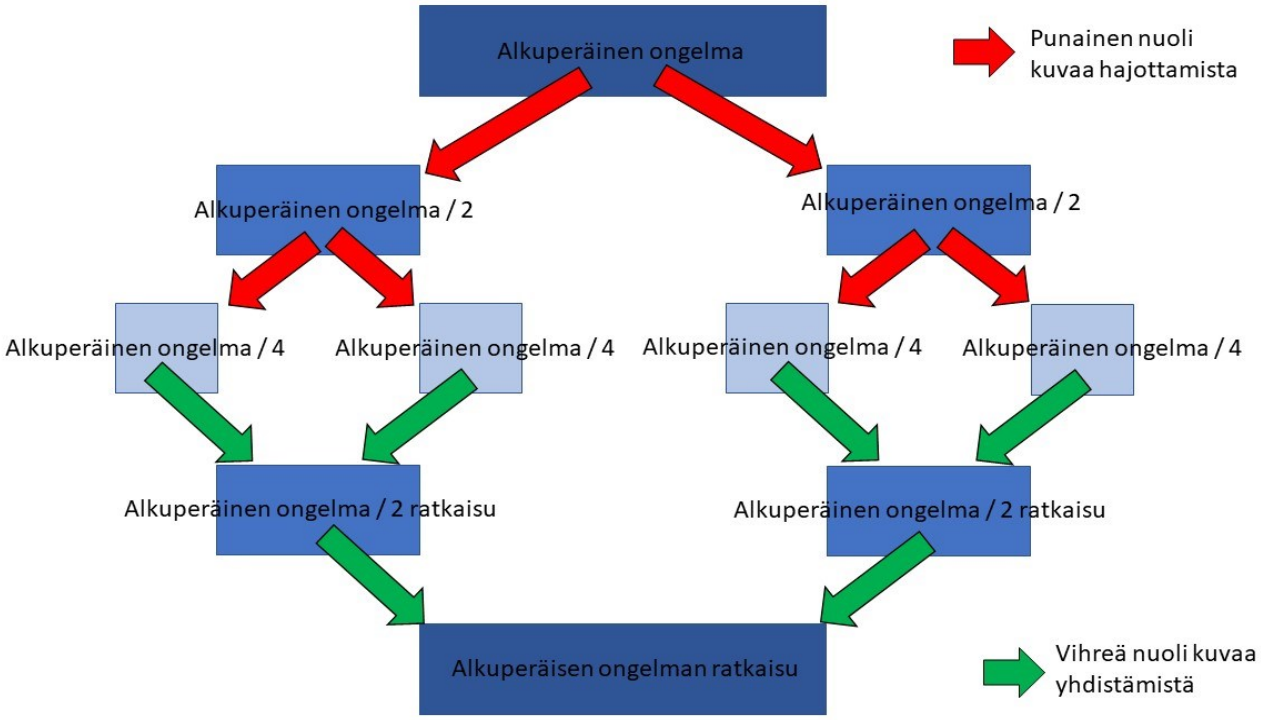
Asymptoottista suoritusaikaa kuvaamaan on käytössä kolme eri merkintää: iso-omega -notaatio(Ω), theetanotaatio(Θ) ja iso-O-notaatio. Iso-omega -notaatio kuvaa kuinka monta operaatioita algoritmi vähintään suorittaa. Tämä notaatio rajaa algoritmin suoritusaikaa vain alhaalta päin. Algoritmista tiedetään ainakin se, että tätä rajaa nopeampi se ei koskaan ole, mutta maksimihitautta algoritmilla ei ole. Theetanotaatio kuvaa kuinka monta operaatiota algoritmi keskimäärin suorittaa. Se rajaa algoritmin suoritusta sekä ylhäältä että alhaalta. Näin ollen algoritmin kasvu pysyy kahden käyrän välissä ja sen hitain ja nopein arvo ovat rajattuja. Iso-O-notaatio kuvaa suurinta mahdollista algoritmin suorittamien operaatioiden määrää. Se rajaa algoritmin ylhäältä ja lupaa, että tätä rajaa hitaampi algoritmi ei voi milloinkaan olla. [2]

2.2 Algoritmien suunnittelu

Algoritmeja suunniteltaessa on ensimmäisenä ymmärrettävä ongelma, joka algoritmin avulla pyritään ratkaisemaan. Suunnitteluvaiheen alkaessa suunnittelijalla on tiedossa algoritmin halutut syötteet ja tulokset. Ongelman ratkaisemiseksi on olemassa neljä eri vaihetta, joita suunnittelija voi käyttää avukseen ongelman ymmärtämisessä. Nämä neljä vaihetta ovat: tutkia miten samankaltaisia ongelmia on ratkaistu aikaisemmin, ymmärtää minkä tyyppiset ratkaisut eivät tuota haluttua tulosta, olla tietoinen siitä mitä resursseja on käytössä sekä tutkia niitä ratkaisuja, joita on aikaisemmin käytetty samankaltaisten ongelmien ratkaisemiseen. Ongelman ratkaisuprosessin aikana nämä neljä vaihetta voivat esiintyä useaan kertaan. [1, luku 2]

Ensimmäinen vaihe algoritmien suunnittelussa on siis ongelman ymmärtäminen. Seuraavana vaiheena on tehdä päätöksiä algoritmin rakenteesta. Näitä päätöksiä ovat esimerkiksi se toteutetaanko rinnakkainen algoritmi vai ei, luodaanko tarkkaan vastauksen palauttava algoritmi vai riittääkö arvioidun ratkaisun palauttava algoritmi, mitä tietorakenteita käytetään ja millä suunnitteluperiaatteella algoritmi luodaan. Seuraavaksi tulee määritellä algoritmi. Algoritmi voidaan määritellä joko pseudokoodin avulla, jonkin olemassa olevan ohjelmointikielen avulla tai flow-kaaviona. Neljäntenä vaiheena tulee tarkastaa, että algoritmi toimii halutulla tavalla. Tämä tarkistus voidaan tehdä syöttämällä algoritmille oikeanlaisia syötteitä ja tarkastamalla, että algoritmi antaa näistä syötteistä halutun tuloksen. Vielä ennen viimeistä vaihetta, joka on algoritmin toteuttaminen, tulee analysoida algoritmin suoritusaikaa. Kannattaa valita toteutus, jolla on pienin suoritusaika. Tämän jälkeen algoritmin voi toteuttaa. [2, luku 1.6]

Monet tehokkaat algoritmit on suunniteltu hajota ja hallitse -suunnitteluperiaatteen pohjalta. Hajota ja hallitse -suunnitteluperiaate hajottaa ensin ongelman kahteen tai useampaan saman suuruiseen osaongelmaan. Tätä vaihetta kutsutaan hajota-vaiheeksi. Osaongelmiin ongelma hajotetaan niin kauan, kunnes osaongelmat ovat tarpeeksi yksinkertaisia ratkaistavaksi sellaisinaan. Osaongelmiin hajotus on toteutettu rekursiivisesti eli algoritmi kutsuu samaa funktiota niin kauan kunnes se kohtaa funktion lopetusehdon. Osaongelmien ratkaisuvaihetta kutsutaan hallitse-vaiheeksi. Hallitse-vaiheen jälkeen tällä suunnitteluperiaatteella toteutetut algoritmit yhdistävät osaongelmat pala kerrallaan alkuperäisen ongelman ratkaisuksi yhdistämisvaiheessa. Hajota ja hallitse -suunnitteluperiaatteella on kolme yllä läpikäytyä vaihetta: hajota, hallitse ja yhdistä. [6, luku 4] Kuvassa 1 on kuvattu hajota ja hallitse -suunnitteluperiaatteen toiminta. Punaiset nuolet kuvaavat hajotettavaa ongelmaa ja vihreät nuolet kuvaavat ongelmien ratkaisuiden yhdistämistä.



Kuva 1: Hajota ja hallitse -suunnitteluperiaatteen toiminta, perustuu lähteeseen [6, luku 4]

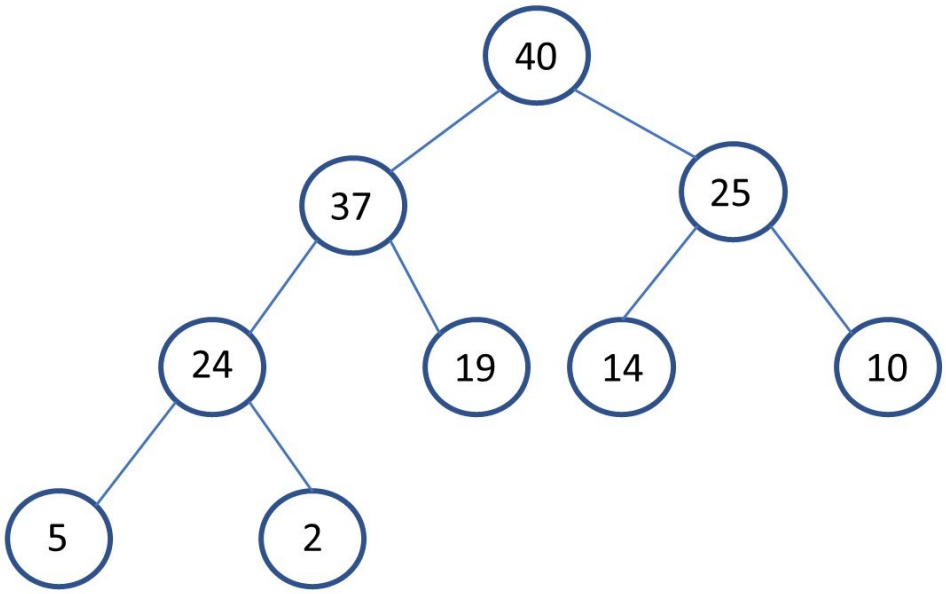
Yksi tärkeimmistä algoritmien suunnitteluperiaatteista on suunnitella algoritmi hyödyntämään saamansa syötettä. Syötettä algoritmin kannattaa hyödyntää siten, että se syöte itsessään tekee jo osan algoritmin työstä. Esimerkiksi, jos tiedetään, että syötteet tulevat aina olemaan merkkijonoja, ei algoritmissa tarvitse olla tarkastusta sille, että syöte on merkkijono. Algoritmin toteutuksessa kannattaa aina käyttää niin yksinkertaisia operaatioita kuin mahdollista. Näin algoritmi saadaan mahdollisimman nopeaksi. Algoritmien toteutuksessa korostuu myös työkalujen ymmärtämisen tärkeys. Sen lisäksi, että suunnittelija tietää, mitä hänen käyttämänsä metodit ja funktiot tekevät, on hänen ymmärrettävä, miten ne toimivat ja mikä on niiden suoritusaika. Tämä on edellytys sille, että suunnittelijan on mahdollista luoda tehokkaita algoritmeja. Se missä ympäristössä algoritmi toimii valmistuttuaan, on tärkeää huomioida suunnitteluvaiheessa. [4]

2.3 Binary-heap tietorakenne

Binary-heap on täydellinen binäärinen hakupuu, jonka juuri solmu on joko listan suurin tai pienin alkio [11]. Binäärinen hakupuu, niin kuin muutkin hakupuut, on hierarkkinen tietorakenne. Binäärinen hakupuu koostuu solmuista, solmujen lapsista, solmujen vanhemmista ja lehdistä. Huippualkiota binäärisessä hakupuussa kutsutaan juureksi. Solmun alapuolella olevia solmuja kutsutaan solmun lapsisolmuiksi ja vastaavasti solmun yläpuolella olevia solmuja kutsutaan vanhemmaksi. Solmu, jolla ei ole yhtään lapsisolmua on lehtisolmu. [12] Täydellisessä binäärihakupuussa mikään muu taso kuin viimeinen ei ole

osittain tyhjä solmuista. Lisäksi viimeisen tason solmujen tulee olla niin vasemmalla kuin mahdollista täydellisessä binäärihakupuussa. [13]

Binary-heap voi olla vain joko minimikeko tai maksimikeko riippuen siitä onko sen juurialkio pienin vai suurin alkio keossa. Jos keko on maksimikeko, tulee kaikkien muiden solmujen olla myös pienempiä kuin niiden vanhempansa. Binary-heap esitetään yleensä listana, jonka i:n solmun vanhempisolmu saadaan lista[(i-1)/2], vasenlapsi lista[(2*1)+1] ja oikealapsi lista[(2*i)+2]. Minimikeon toiminnallisuudet ovat samanlaiset kuin maksimikeolla. [11] Kuvassa kaksi on havainnollistettu maksimikekoa.



Kuva 2: Binäärinen maksimikeko

Kuvassa 2 juurisolmuna on solmunumero 40. Sen vasempana lapsena on solmu 37 ja oikeana lapsena solmu 25. Solmu 40 on solmujen 37 ja 25 vanhempi. Solmu 5 on esimerkiksi lehtisolmu, koska sillä ei ole yhtään lapsisolmua. Kuvan binary-heap on listamuodossa esitettynä: 40,37,25,24,19,14,10,5,2. Kuvasta nähdään, että vain viimeisen tason solmut ovat kaikki mahdollisimman vasemmalla ja taso on vain osittain täysi. Kuvan 2 keon muut tasot ovat täynnä solmuja. Kyseessä on siis täydellinen binäärinen hakupuu [11].

3. VERTAILTAVAT JÄRJESTYSALGORITMIT

Koska monet algoritmit tarvitsevat saamansa syötteen oikeassa järjestyksessä, on järjestysalgoritmien rooli merkittävä tietotekniikassa. Monet algoritmit myös järjestävät syötteensä välillä moneenkin kertaan algoritmin suorituksen aikana. Koska järjestämisen tarve on suuri ja erilainen erilaisissa tapauksissa, on järjestysalgoritmeja monia erilaisia. Se mikä järjestysalgoritmi on paras, riippuu monesta asiasta, kuten siitä kuinka suuri on järjestettävä lista. [3] Tässä luvussa esitellään kolme eri järjestysalgoritmia: quicksort luvussa 3.1, insertion sort luvussa 3.2 ja heap sort luvussa 3.3. Nämä kolme algoritmia on valittu, koska ne ovat laajalti käytössä, soveltuvat erilaisiin käyttötarkoituksiin ja kukin on toteutettu eri tavalla.

3.1 Quicksort

Quicksort toimii hajota ja hallitse -toimintaperiaatteella. Quicksort valitsee pivot-alkion ja jakaa parametrinaan saamaansa listan osiin pivot-alkion kohdalta. Pivot-alkio on alkio, joka yhdistää lajiteltavan vektorin erilaiset alkiojoukot. Alkioiden jakaminen tehdään partition-funktion avulla. Partition-funktion idea on jakaa saamansa lista pivot-alkion kohdalta, siten että pivot-alkiota suuremmat alkiot siirtyvät pivot-alkion oikealle puolelle ja pivot-alkiota pienemmät alkiot siirtyvät sen vasemmalle puolelle. Quicksort kutsuu itseään rekursiivisesti niin kauan, kunnes lista on järjestyksessä. [7]

Ohjelmassa 1 on kuvattu quicksort-algoritmin toiminta. Quicksort saa parametreinään lajiteltavan vektorin sekä lajittelun aloitus- ja lopetuspisteet. Funktio kutsuu partition-funktiota, jos sen aloituspiste on suurempi kuin lopetuspiste. Partition-funktio palauttaa pivot-alkion kohdan. Tämä alkio on jo oikealla kohdalla järjestettävässä listassa ja yhdistää oikean- ja vasemmanpuoleisen alalistan. Riveillä kuusi ja seitsemän quicksort kutsuu itseään rekursiivisesti lajitellakseen pivot-alkion jakaman oikean- (rivi 6) ja vasemmanpuoleisen (rivi 7) alalistan.

```
void Quicksort(std::vector<int>& vectorToSort, int startPoint, int endpoint) {
2       if (startPoint < endpoint) {
                // Alkion indeksi, mikä on jo oikealla paikalla.
4               int pivotPoint = partition(vectorToSort, startPoint, endpoint);
                // Lajitellaan vasen- ja oikea-alalista.
6               quickSort(vectorToSort, startPoint, pivotPoint - 1);
                quickSort(vectorToSort, pivotPoint + 1, endPoint);
8       }
}
```

Ohjelma 1: Quicksort algortimin toteutus C++-koodilla. Perustuu lähteeseen [7].

Ohjelmassa 2 on kuvattu partition-funktion toiminta. Funktio saa parametrinaan jaettavan listan, aloituspisteen ja lopetuspisteen. Pivot-alkioksi valitaan parametrina saadun funktion ensimmäinen arvo. Rivillä 4 funktio alustaa molemmat alalistat tyhjäksi. Riveillä 6-12 funktio käy listan läpi aloittaen pivot-alkiota seuraavasta alkiosta. Joka kierroksella funktio vertaa käsiteltävää alkiota pivot-alkioon. Jos se on pienempi kuin pivot-alkio, se siirretään vasempaan alalistaan. Muussa tapauksessa ei tehdä mitään, jolloin käsiteltävä alkio jää oikeaan alalistaan. Lopuksi rivillä 14 siirretään pivot-alkio oikeaan kohtaan alkuperäisessä listassa ja rivillä 15 palautetaan pivot-alkion sijainti.

```
int partition(std::vector<int>& myVector, int startPoint, int endPoint){
2       int pivotPoint = myVector.at(startPoint);
        // Koska molemmat alalistat alustetaan tyhjiksi
4       int middlePoint = startPoint;
        int round = startPoint + 1;
6       while (round <= endPoint) {
                // Käsiteltävä alkio on vasemmassa alalistassa.
8               if (myVector.at(round) < pivotPoint) {
                        ++ middlePoint;
8                       // vaihdetaan middlePoint vasempaan alalistaan.
                        std::swap(myVector.at(round), myVector.at(middlePoint));
10              }
                ++round;
12      }
        // Siirretään pivot-alkio oikeaan kohtaan
14      std::swap(myVector.at(startPoint), myVector.at(middlePoint));
        return middlePoint; // pivot-alkion indeksi
}
```

Ohjelma 2: Quicksortin käyttämän partition-funktion totetus C++-kielellä, perustuu lähteeseen [7].

Quicksortin paras ja keskimääräinen suoritusaika on Θ(nlogn), kun se lajittelee n-alkion pituista listaa. Quicksortin toteutuksessa on vain vähän koodia, jonka suoritusaika on Θ(nlogn). Näin ollen quicksort on erittäin tehokas algoritmi. Tehokkuutensa ansiosta quicksort on usein paras vaihtoehto järjestysalgoritmeja valittaessa. Pahimman tapauksen suoritusaika quicksortilla on Ω(n^2). [3, luku 7] Quicksortin pahin tapaus toteutuu, kun järjestettävän listan alkiot ovat täysin vastakkaisessa järjestyksessä kuin järjestetyssä listassa. Quicksort saavuttaa parhaan tapauksen täysin järjestyksessä olevalla listalla, melkein järjestetyllä listalla tai listalla, jossa alkiot ovat satunnaisessa järjestyksessä. [15] Koska quicksort ei pidä saman arvon omaavien alkioiden järjestystä samana kuin se alun perin oli, niin quicksort on epävakaa järjestysalgoritmi [8].

3.2 Insertion sort

Insertion sort toimii valitsemalla ensin toisena olevan alkion listasta ja vertaamalla sitä ensimmäiseen alkioon. Jos toinen alkio on suurempi kuin ensimmäinen alkio, toinen alkio jää paikalleen. Jos taas ei, niin toinen alkio siirtyy ensimmäisen alkion eteen. Seuraavaksi

insertion sort valitsee kolmannen alkion ja vertaa sitä ensin toiseen alkioon. Jos se on pienempi kuin toinen alkio, vaihtavat kolmas alkio ja toinen alkio paikkoja. Sitten toisena olevaa alkiota verrataan ensimmäiseen alkioon ja tehdään tarvittava siirto. Ja näin jatkuu kunnes lista on kokonaan järjestyksessä. Insertion sort käy listan jokaisen alkion läpi ja vertaa niitä sen hetkisen alkion vasemmalla puolella oleviin alkioihin, kunnes löytää alkion oikean paikan. [14]

Ohjelmassa 3 on kuvattu insertion_sort-funktion toiminta. Funktio saa parametrinaan listan, jonka se järjestää. Funktio käy listan lävitse ja jokaisella kierroksella ensimmäiseksi alustaa nykyisen arvon listan j:nnellä alkiolla sekä luo apumuuttujan i. Rivejä 5-8 funktio suorittaa niin kauan, kun i on suurempaa kuin nolla ja listan i:nnes arvo on suurempaa kuin nykyinen arvo. Rivillä 6 listan i:nnes ja sitä seuraava alkio vaihtavat paikkaansa. Jokaisen kierroksen lopuksi i:stä seuraavaan alkion arvoksi asetetaan nykyinen arvo.

```
void insertion_sort(std::vector<int>& vectorToSort) {

2       for (int j = 1; j < vectorToSort.size(); j++) {

                int current_value = vectorToSort.at(j);

4               int i = j-1;

                while (i>0 && vectorToSort.at(i) > current_value) {

6                       vectorToSort.at(i+1) = vectorToSort.at(i);

                        i = i-1;

8               }

                vectorToSort.at(i+1) = current_value;

10      }

}
```

Ohjelma 3: Funktion insertion_sort toteutus, perustuu lähteeseen [3, luku 2].

Insertion sort saavuttaa parhaan suoritusaikansa O(n) melkein ja täysin järjestetyillä listoilla. Pahimmassa tapauksessa insertion sortin suoritusaika on Ω(n^2). Insertion sortin pahin tapaus toteutuu, kun järjestettävän listan alkiot ovat täysin vastakkaisessa järjestyksessä kuin järjestetyssä listassa [15]. Vaikka insertion sortin paras suoritusaika onkin O(n), niin silti sen keskimääräinen suoritusaika on Θ(n^2) [10] Insertion sort sopii pienten sekä melkein ja täysin järjestyksessä olevien listojen lajitteluun, jolloin sen suoritusaika on O(n). Pienten listojen lajitteluun insertion sort soveltuu, koska itse algoritmi on lyhyt, jolloin lajiteltaville alkioille ei suoriteta montaa operaatiota. [3, luku 2] Insertion sort on vakaa algoritmi eli sen järjestäessä alkiota, saman suureiset alkiot pysyvät samassa järjestyksessä kuin ne olivat järjestettäväksi tulleessa listassa [15].

3.3 Heap sort

Heap sortin suunnitteluperiaatteena on tietorakenteen käyttäminen tiedon hallintaan. Heap sort käyttää tietorakenteenaan kekoa. Heap sort luo ensin maksimikeon, jonka jälkeen käy järjestettävän listan lävitse vaihtamalla jokaisella kierroksella listan ensimmäisen alkion ja läpikäytävän alkion paikkoja. Lisäksi funktio kutsuu jokaisella kierroksella funktiota max_heapify säilyttääkseen kekorakenteen maksimikekona koko lajittelun ajan. [3, luku 6] Heap sort on mahdollista toteuttaa myös minimikeon avulla.

Ohjelmassa 4 on kuvattu max_heapify-funktion toiminta. Max_heapify saa parametreinaan lajiteltavan listan, indeksin ja keon koon. Funktio asettaa saamansa indeksin suurimmaksi ja laskee indeksin avulla vasemman- ja oikeanlapsenindeksit. Rivillä 5 funktio tarkastaa, onko indeksin vasemmanlapsen indeksi pienempi kuin keon koko ja onko listanarvo vasemmanlapsen kohdalla suurempi kuin listanarvo suurimman kohdalla, jos on, asetetaan vasemmanlapsen indeksi suurimmaksi. Vastaava tarkastelu tehdään oikeanlapsen indeksille rivillä 9. Tämän jälkeen, jos suurin arvo ei ole sama kuin indeksi, niin listassa indeksin kohdalla oleva alkio vaihtaa paikkaa suurimman kohdalla olevan alkion kanssa ja funktio kutsuu itseään rekursiivisesti.

```
void max_heapify(std::vector<int>& vectorToHeap, int index, int heap_size) {
2       int largest = index;
        int leftChild = 2*index + 1;
4       int rightChild = 2*index + 2;
        if (leftChild < heap_size && vectorToHeap.at(leftChild) > vectorToHeap.at(largest))
6       {
                largest = leftChild;
8       }
        if (rightChild < heap_size && vectorToHeap.at(rightChild) >
10                                                  vectorToHeap.at(largest)) {
                largest = rightChild;
12      }
        if (largest != index) {
14              std::swap(vectorToHeap.at(index), vectorToHeap.at(largest));
                max_heapify(vectorToHeap, largest, heap_size);
16      }
}
```

Ohjelma 4: Funktion max_heapify toteutus. perustuu lähteeseen [9]

Ohjelmassa 5 on kuvattu build_max_heap funktion toimintaa. Funktio rakentaa maksimikeon. Build_max_heap saa parametrinaan sen listan, josta se rakentaa maksimikeon. Funktio kutsuu max_heapify funktiota lista/2-1 kertaa järjestäen listan alkiot maksimikeoksi.

```
void build_max_heap(std::vector<int>& vectorToHeap) {
2       for (int round = (vectorToHeap.size()/2)-1; round >= 0; round--) {
                max_heapify(vectorToHeap, round, vectorToHeap.size());
4       }
}
```

Ohjelma 5: Funktion build_max_heap toteutus, perustuu lähteeseen [9]

Ohjelmassa 6 on kuvattu heap_sort-funktion toiminta. Funktio saa parametrinaan listan, jonka se järjestää. Ensin funktio rakentaa maksimikeon kutsumalla build_max_heap -funktiota rivillä 2. Funktio käy läpi kaikki listan alkiot ja vaihtaa jokaisen alkion paikkaa saamansa listan ensimmäisen alkion kanssa, kunnes kaikki alkiot on käyty läpi. Lisäksi jokaisella kierroksella funktio kutsuu max_heapify-funktiota, jotta lista pysyy maksimikekona.

```
void heap_sort(std::vector<int>& vectorToHeap) {
2       build_max_heap(vectorToHeap);
        for (int i = vectorToHeap.size()-1; i >= 0; i--) {
4               std::swap(vectorToHeap.at(0), vectorToHeap.at(i));
                max_heapify(vectorToHeap, 0, i);
6       }
}
```

Ohjelma 6: Funktion heap_sort toteutus, perustuu lähteeseen [9]

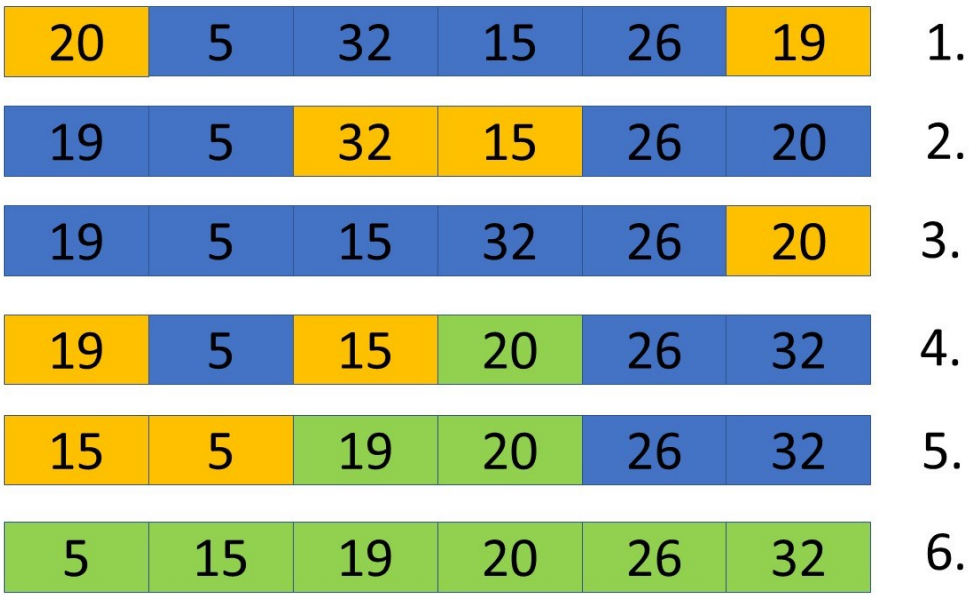
Heap sortin keskimääräinen suoritusaika on Θ(nlogn). Myös parhaan ja huonoimman tapauksen suoritusaika on sama kuin keskimääräinen suoritusaika. [3] Heap sort saavuttaa parhaimman suoritusajan lähes järjestyksessä olevilla listoilla. Pahin tapaus toteutuu, kun lista on satunnaisessa järjestyksessä. [15] Heap sortilla ei ole kovin montaa sovellusaluetta, vaikka sen suoritusaika on huonoimmillaankin Ω(nlogn). Yleisemmin heap sortia käytetään järjestelmissä, joissa ollaan tarkkoja turvallisuudesta ja rinnakkaisuudesta. [9] Heap sort on epävakaa algoritmi, sillä se ei pidä saman arvon omaavia alkioita alkuperäisessä järjestyksessä [8].

4. JÄRJESTYSALGORITMIEN VERTAILU

Tässä luvussa vertaillaan quicksortin, insertion sortin ja heap sortin ominaisuuksia toisiinsa. Luku 4.1 keskittyy vertailemaan näiden algoritmien toimintalogiikkaa toisiinsa. Toimintalogiikkaa vertaillaan, koska algoritmeja käyttäessä on hyvä tietää, miten ne oikeasti toimivat, jotta osaa valita itselleen sopivan järjestysalgoritmin. Luvussa 4.2 vertaillaan algoritmien asymptoottista tehokkuutta parhaassa, keskimääräisessä ja pahimmassa tapauksessa. Tämä vertailu tehdään, jotta nähdään miten tehokkaita algoritmit ovat toisiinsa verrattuina. Luvun 4 viimeisessä luvussa keskitytään vertailemaan mihin käyttötarkoitukseen algoritmit soveltuvat, millä suunnitteluperiaatteella algoritmit on toteutettu sekä ovatko algoritmit vakaita vai eivät.

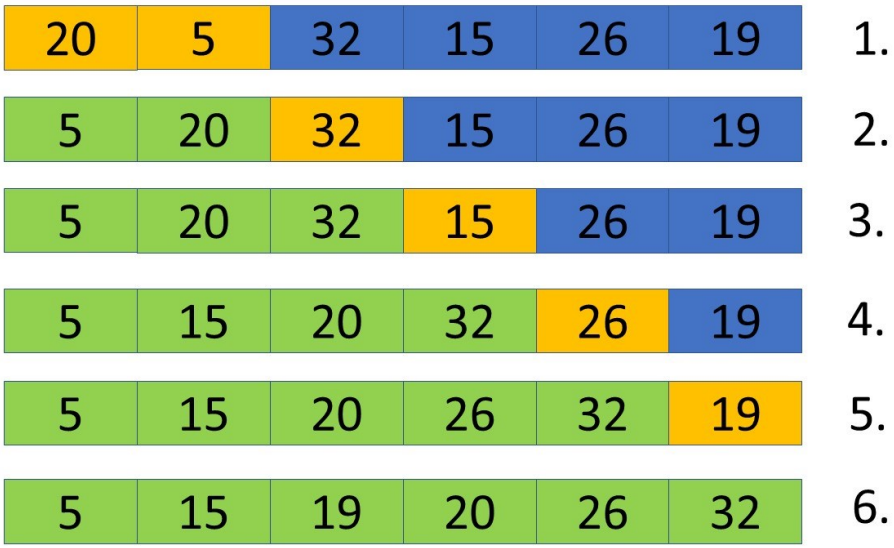
4.1 Algoritmien toimintalogiikan vertailu

Quicksortin, insertion sortin ja heap sortin toimintalogiikat eroavat huomattavasti toisistaan. Quicksort toimii hajota ja hallitse -periaatteella eli ensin se hajottaa lajiteltavan listan pieniin paloihin ja sitten kokoaa ne haluttuun järjestykseen. Insertion sort lisää alkion aina oikealle paikalle listassa ennen kuin siirtyy seuraavaan alkioon. Heap sort eroaa quicksortista ja insertion sortista siten, että siinä algoritmi rakentaa itse kekotietorakenteen, jota käyttää lajittelussa. Kuvissa 3, 4 ja 5 on oranssilla merkitty ne alkiot, jotka vaihtavat paikkoja keskenään, sinisellä alkiot, jotka eivät ole vielä oikealla paikalla ja vihreällä oikealla paikalla olevat alkiot.



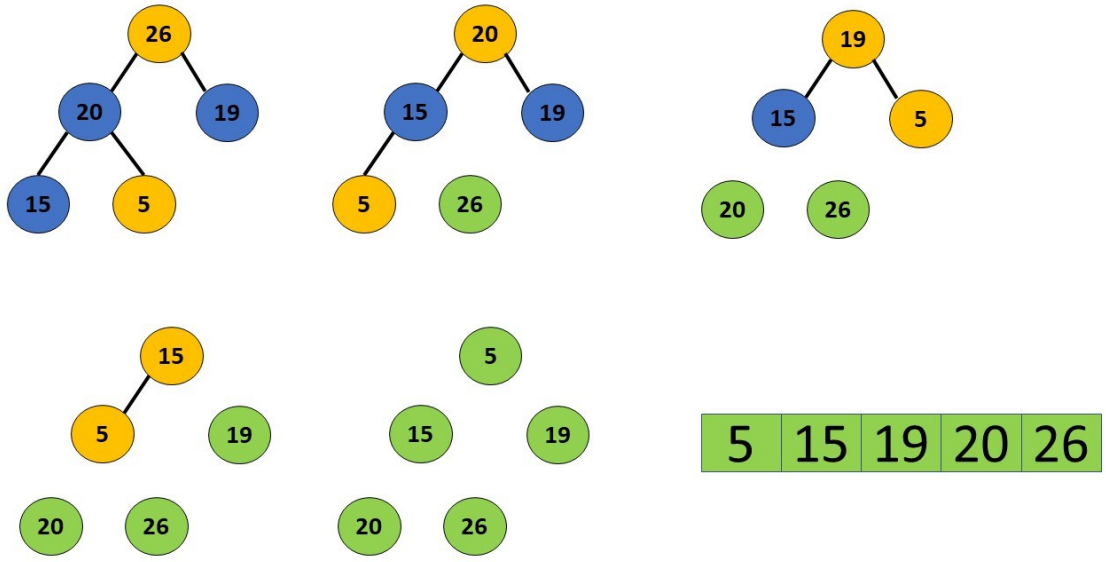
Kuva 3: Quicksortin toimintalogiikka, perustuu lähteisiin [3, luku 7][7].

Kuvasta 3 nähdään, miten lista järjestyy quicksorttia käyttäen. Quicksort ja insertion sort järjestävät molemmat listoja haluttuihin järjestyksiin. Kuvassa 4 näkyy listan järjestäminen insertion sortin avulla. Kuvista nähdään, että insertion sortin edetessä enemmän alkioita on oikeassa järjestyksessä nopeammin kuin quicksortin edetessä. Insertion sort aloittaa järjestämisen toisesta alkiosta ja etenee tasaisesti listan loppuun asti. Se vertaa aina seuraavaa alkiota vasemmalla puolella järjestyksessä oleviin alkioihin ja lisää alkion oikeaan kohtaan listassa. [14] Quicksort puolestaan valitsee pivot-alkion. Pivot-alkioksi valitaan yleensä aina ensimmäinen alkio. Tämän jälkeen quick sort vertailee alkioita pivot alkioon jakaakseen listan kahteen osaan. Seuraavaksi sama toistetaan molemmille osille ja taas uusille osille, kunnes quicksort on jakanut listan tarpeeksi pieniin paloihin. Lopuksi quicksort yhdistää palat saaden alkuperäisen listan järjestykseen. [7]



Kuva 4: Insertion sortin toimintalogiikka. Perustuu lähteeseen [14].

Siinä missä quicksort ja insertion sort järjestävät listoja järjestykseen, niin heap sort järjestää kekoa. Heap sort ottaa parametrinaan järjestettävän listan, mutta luo siitä joko maksimikeon tai minimikeon ennen kuin rupeaa järjestämään sitä [3, luku 6]. Tämän takia heap sort eroaa toteutukseltaan paljon quicksortista ja insertion sortista. Kuvassa 5 on kuvattu heap sortin toimintalogiikkaa. Ensin heap sort luo maksimikeon (voisi luoda minimikeonkin) ja sitten siirtää juurialkion viimeiseksi samalla poistaen sen keosta. Tämän jälkeen heap sort palauttaa maksimikeon ja siirtää taas juurialkion viimeiseksi ja poistaa sen keosta. Näin jatkuu, kunnes kaikki keon alkiot on käyty läpi. Kun alkiot on käyty läpi, parametrina saatu lista on järjestyksessä. [3, luku 6]



Kuva 5: Heap sortin toimintalogiikka, perustuu lähteeseen [3, luku 6]

Insertion sortilla on näistä kolmesta järjestysalgoritmista kaikki yksinkertaisin toteutus. Se on toteutettu kahden sisäkkäisen toistorakenteen avulla [3, luku 2]. Koska algoritmit on toteutettu eri tavoilla, niin niiden toteutuksessa on käytetty eriävä määrä funktioita. Quicksort on toteutettu käyttäen kahta eri funktiota, heap sort kolmea ja insertion sort on kokonaisuudessaan vain yksi funktio.

4.2 Asymptoottisen tehokkuuden vertailu

Jokaisella järjestysalgoritmilla on omat parhaimman, keskimääräisen ja pahimman tapauksen suoritusaikansa. Algoritmeja suunnitellessa tulee suoritusajat ottaa huomioon, koska ne vaikuttavat siihen miten tehokkaita algoritmit ovat. Quicksortin, insertion sortin ja heap sortin parhaan, keskimääräisen ja pahimman tapauksen suoritusajat eroavat toisistaan välillä aika paljonkin. Esimerkiksi suoritusaika O(nlog) tarkoittaa, että algoritmi suorittaa parhaassa tapauksessa nlogn määrän operaatioita, missä n on järjestettävien alkioiden määrä. Taulukoissa 1, 2 ja 3 suoritusaikoja on tutkittu seuraavilla n:n arvoilla: 100, 1 000, 10 000, 100 000, 1 000 000, 10 000 000, 100 000 000. Tutkittavien alkioiden määrä on valittu, siten että ne kertaantuvat aina kymmenellä edelliseen nähden, jotta saadaan mahdollisimman kattava vertailuaineisto.

Taulukossa 1 on esitelty quicksortin, insertion sortin ja heap sortin parhaimman tapauksen asymptoottisia suoritusaikoja kolmen numeron tarkkuudella. Näistä kolmesta algoritmista insertion sortilla on paras parhaan tapauksen suoritusaika, joka on O(n). Alkioiden määrän kasvaessa nähdään, että insertion sortin tekemien operaatioiden määrä jää huomattavasti alle sen, mitä quicksortin ja heap sortin tekemien operaatioiden määrä. Jo sadannen tuhannen alkion kohdalla insertion sort suorittaa kymmenen kertaa vähemmän

operaatioita kuin quicksort ja heap sort, joiden parhaimman tapauksen suoritusaika on O(nlogn).

Taulukko 1: Quicksortin, insertion sortin ja heap sortin parhaan tapauksen asymptoottisten suoritusaaikojen vertailutaulukko.

Parhaan tapauksen asymptoottinen suoritusaika (kolmen numeron tarkkuudella)			
n	Quicksort O(nlogn)	Insertion sort O(n)	Heap sort O(nlogn)
100	461	100	461
1 000	6 910	1 000	6 910
10 000	92 100	10 000	92 100
100 000	1 150 000	100 000	1 150 000
1 000 000	13 800 000	1 000 000	13 800 000
10 000 000	161 000 000	10 000 000	161 000 000
100 000 000	1 840 000 000	100 000 000	1 840 000 000

Taulukosta 2 nähdään, että insertion sortin keskimääräinen suoritusaika on näistä kolmesta algoritmista huonoin. Toisin kuin insertion sortilla, quicksortilla ja heap sortilla on samat parhaimman ja keskimääräisen tapauksen asymptoottiset suoritusajat. Quicksortin ja heap sortin keskimääräinen asymptoottinen suoritusaika on Θ(nlogn). Quicksort ja heap sort suorittavat alle kymmenen tuhannen alkion määrillä keskimäärin 100 kertaa vähemmän operaatioita kuin insertion sort. Alkioiden määrän noustessa yli kymmeneen tuhanteen ero alkaa olla huomattavasti suurempi. Jo miljoonan alkion kohdalla insertion sort suorittaa keskimäärin satatuhatta kertaisen määrän operaatioita quicksortiin ja heap sortiin verrattuna.

Taulukko 2: Quicksortin, insertion sortin ja heap sortin keskimääräisen tapauksen asymptoottisten suoritusaikojen vertailutaulukko.

Keskimääräinen asymptoottinen suoritusaika (kolmen numeron tarkkuudella)			
n	Quicksort Θ(nlogn)	Insertion sort Θ(n^2)	Heap sort Θ(nlogn)
100	461	10 000	461
1 000	6 910	100 000	6 910
10 000	92 100	100 000 000	92 100
100 000	1 150 000	1,00^10	1 150 000
1 000 000	13 800 000	1,00^12	13 800 000
10 000 000	161 000 000	1,00^14	161 000 000
100 000 000	1 840 000 000	1,00^16	1 840 000 000

Taulukosta kolme nähdään, että quicksortilla ja insertion sortilla on sama pahimman tapauksen asymptoottinen suoritusaika Ω(n^2). Heap sortilla myös pahimman tapauksen asymptoottinen suoritusaika on Ω(nlogn). Quicksort ja insertion sort suorittavat alle kymmenen tuhannen alkion määrillä keskimäärin 100 kertaa enemmän operaatioita kuin heap sort. Alkioiden määrän noustessa yli kymmeneen tuhanteen ero alkaa olla huomattavasti suurempi. Jo miljoonan alkion kohdalla quicksort ja insertion sort suorittavat keskimäärin satatuhatta kertaisen määrän operaatioita heap sortiin verrattuna.

Taulukko 3: Quicksortin, insertion sortin ja heap sortin pahimman tapauksen asymptoottisten suoritusaikojen vertailutaulukko.

Pahimman tapauksen asymptoottinen suoritusaika (kolmen numeron tarkkuudella)			
n	Quicksort Ω(n^2)	Insertion sort Ω(n^2)	Heap sort Ω(nlogn)
100	10 000	10 000	461
1 000	100 000	100 000	6 910
10 000	100 000 000	100 000 000	92 100
100 000	1,00^10	1,00^10	1 150 000
1 000 000	1,00^12	1,00^12	13 800 000
10 000 000	1,00^14	1,00^14	161 000 000
100 000 000	1,00^16	1,00^16	1 840 000 000

Taulukoista 1, 2 ja 3 nähdään, että vaikka insertion sortilla on paras parhaimman tapauksen asymptoottinen suoritusaika, se ei silti ole hyvä valinta suurien alkiomäärien järjestämiseen, koska sen keskimääräinen asymptoottinen suoritusaika on sama kuin sen pahimman tapauksen. Huomataan myös, että heap sortin asymptoottinen suoritusaika on Θ(nlogn) parhaassa, keskimääräisessä ja pahimmassa tapauksessa. Taulukoista käy ilmi, että asymptoottisen suoritusajan noustessa algoritmin suoritettavien operaatioiden määrä lisääntyy todella paljon. Esimerkiksi siirryttäessä Θ(nlogn) asymptoottisesta tehokkuudesta Θ(n^2):een, algoritmin suoritettavien operaatioiden määrä kasvaa noin sadallatuhannella, kun algoritmi käsittelee miljoonaa alkiota kerralla.

4.3 Muiden ominaisuuksien vertailu

Taulukossa 4 on vertailtu järjestysalgoritmien suunnitteluperiaatteita, sitä mihin algoritmit parhaiten soveltuvat sekä ovatko algoritmit vakaita vai eivät. Taulukosta nähdään, että quicksort ja heap sort ovat molemmat epävakaita algoritmeja. Nämä algoritmit eivät pidä listan alkioita, joilla on sama arvo, samassa järjestyksessä kuin ne olivat algoritmille järjestettäväksi tulleessa listassa. Insertion sort säilyttää alkioiden järjestyksen samana kuin järjestettäväksi saamassaan listassa, joten se on vakaa algoritmi.

Taulukko 4: Järjestysalgoritmien suunnitteluperiaatteen, sovellusalueen ja vakauden vertailu.

Järjestys algoritmi	Suunnitteluperiaate	Soveltuu parhaiten	Algoritmin vakaus
Quicksort	Hajota ja hallitse [7]	Suurimpaan osaan tapauksista. Paras järjesgtysalgoritmi tehokkuutensa takia. [3, luku 7]	Epävakaa [8]
Insertion sort	Kaksi for-silmukkaa [3, luku 2]	Pienten sekä melkein ja täysin järjestyksessä olevien listojen lajitteluun [3, luku 2].	Vakaa [15]
Heap sort	Tietorakenne [3, luku 6]	Järjestelmiin, joissa ollaan tarkkoja turvallisuudesta ja rinnakkaisuudesta [9].	Epävakaa [8]

Quicksort, insertion sort ja heap sort on kaikki toteutettu eri suunnitteluperiaatteella. Kuten taulukosta voidaan nähdä, niin quicksort edustaa hajota ja hallitse –suunnitteluperiaatetta, insertion sort on toteutettu kahden for-silmukan avulla ja heap sort luo uuden tietorakenteen, johon järjestämisen ajaksi laittaa järjestettävänä saamansa listan alkiot ja sitten järjestää tietorakenteen. Taulukosta nähdään, että quicksort, insertion sort ja heap sort soveltuvat kaikki hieman erilaisiin käyttötapauksiin. Quicksortia voidaan käyttää laajasti erilaisissä listojen järjestystä vaativissa tapauksisissa ja se on tehokkuutensa ansiosta paras järjestysalgoritmi [3, luku 7]. Insertion sort puolestaan soveltuu hyvin pienten sekä melkein ja täysin järjestyksessä olevien listojen lajitteluun [3, luku 2]. Heap sortia ei käytetä kovin monella sovellusalueella ja useimmiten sitä käytetään järjestelmissä, joissa ollaan tarkkoja turvallisuudesta ja rinnakkaisuudesta [9].

5. YHTEENVETO

Tässä työssä vertailtiin kolmea eri järjestysalgoritmia toisiinsa. Nämä algoritmit olivat quicksort, insertion sort ja heap sort. Työn tavoitteena oli vastata kysymykseen mitkä ovat quicksortin, insertion sortin ja heap sortin suurimmat erot algoritmien toteutustasolla. Quicksort on paras järjestysalgoritmi tehokkuutensa takia ja sitä käytetään siksi todella paljon. Insertion sortia käytetään pienien ja melkein sekä täysin järjestyksessä olevien listojen järjestämiseen ja heap sortia käytetään järjestelmissä, joissa ollaan tarkkoja turvallisuudesta ja rinnakkaisuudesta.

Quicksortin, insertion sortin ja heap sortin toimintalogiikat eroavat toisistaan huomattavasti. Quicksort toimii hajoita ja hallitse –suunnitteluperiaatteella eli se hajottaa lajiteltavan listan paloiksi ja sitten kokoaa sen oikeaan järjestykseen. Insertion sort puolestaan vertaa aina seuraavaa alkiota vasemmalla puolella järjestyksessä oleviin alkioihin ja lisää alkion oikeaan kohtaan listassa. Siinä missä quicksort ja insertion sort järjestävät listoja, heap sort luo parametrinaan saamastaan listasta keon ja järjestää sen. Quicksort ja heap sort ovat molemmat epävakaita järjestysalgoritmeja ja insertion sort on vakaa järjestysalgoritmi.

Tehdyssä vertailussa selvisi, että insertion sortilla on paras parhaan tapauksen suoritusaika, mutta huonoin keskimääräinen suoritusaika. Heap sortin suoritusaika on sama kaikissa kolmessa tapauksessa (nlogn), ja quicksortin suoritusaika on nlogn parhaassa ja keskimääräisessä tapauksessa ja n^2 pahimmassa tapauksessa. Quicksortin ja insertion sortin pahin tapaus toteutuu, kun järjestettävän listan alkiot ovat täysin vastakkaisessa järjestyksessä kuin järjestetyssä listassa. Heap sort saavuttaa pahimman tapauksen suoritusaikansa puolestaan satunnaisessa järjestyksessä olevilla listoilla.

Lähteet

[1] P. Mueller, L.Massaron, Algorithms for Dummies, John Wiley & Sons, 2017. Saatavissa (viitattu 14.09.2018): http://library.books24x7.com.libproxy.tut.fi/toc.aspx?bookid=125664

[2] S. Singhal, Analysis and Design of Algorithms: A Beginner's Hope, BPB Publications, 2018. Saatavissa (viitattu 14.09.2018): http://library.books24x7.com.libproxy.tut.fi/toc.aspx?bkid=138882

[3] T. Cormen, C. Leiserson, R. Riverst, C. Stein, Introduction to Algorithms, Third Edition, The MIT Press, 2009. Saatavissa (viitattu 14.09.2018): http://library.books24x7.com.libproxy.tut.fi/toc.aspx?bkid=49924

[4] J. Cutrell, Understanding the Principles of Algortihm Design, 2012. Saatavissa (viitattu 14.09.2018): https://code.tutsplus.com/tutorials/understanding-the-principles-of-algorithm-design--net-26561

[5] T. Cormen, D. Balkcom, Asymptotic notation, Khan Academy. Saatavissa (Viitattu 14.09.2018): https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

[6] S. Bajaj Mundra, Design and Analysis of Algorithms, Laxmi Publications, 2013. Saatavissa (Viitattu 28.09.2018): http://library.books24x7.com.libproxy.tut.fi/assetviewer.aspx?bookid=62006&chunkid=852968068&rowid=158&noteMenuToggle=0&leftMenuState=1

[7] GeeksforGeeks, C++ Program for Quick Sort. Saatavissa (Viitattu 11.10.2018): https://www.geeksforgeeks.org/cpp-program-for-quicksort/

[8] Stack Overflow, What is stability in sorting algorithms and why it is important. Saatavissa (Viitattu 16.10.2018): https://stackoverflow.com/questions/1517793/what-is-stability-in-sorting-algorithms-and-why-is-it-important

[9] Programiz, Heap Sort Algorithm. Saatavissa (Viitattu 16.10.2018): https://www.programiz.com/dsa/heap-sort

[10] Khan Academy, Analysis of insertion sort. Saatavissa (Viitattu 16.10.2018): https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort

[11] GeegsforGeeks, Binary Heap. Saatavissa (Viitattu 23.10.2018): https://www.geeksforgeeks.org/binary-heap/

[12] GeegsforGeeks, Binary Tree, set 1 (Introduction). Saatavissa (Viitattu 23.10.2018): https://www.geeksforgeeks.org/binary-tree-set-1-introduction/

[13] GeegsforGeeks, Binary Tree, set 3 (Types of Binary Tree). Saatavissa (Viitattu 23.10.2018): https://www.geeksforgeeks.org/binary-heap/

[14] Programiz, Insertion Sort Algorithm. Saatavissa (Viitattu 06.11.2018): https://www.programiz.com/dsa/insertion-sort

[15] N. Faujdar, S. Prakash Ghrera, Analysis and Testing of Sorting Algorithms on a Standard Dataset, IEEE, 2015. Saatavissa (viitattu 17.11.2018): https://ieeexplore.ieee.org/document/7280062/authors#authors